Proxy Re-Encryption Protocol

IronCore Labs

July 10, 2018 – Version 1.1

Executive Summary

Synopsis

From February 26 to March 18, 2018, IronCore Labs engaged NCC Group's Cryptographic Services Practice to perform a review of their proxy re-encryption protocol and implementation. The review aimed at validating the specific choice of the pairing-friendly elliptic curve in the protocol, and verifying that the Scala implementation is a secure incarnation of that protocol. The Scala code targets both the Java virtual machine, and in-browser execution through Scala-js; this translates Scala code to JavaScript or WebAssembly.

The review covered IronCore's `recrypt` library implementation, using internal version numbers 7.0.2-SNAPSHOT, then 8.0.1-SNAPSHOT, and finally 11.0.0-SNAPSHOT. This last version was verified to be strictly equivalent to the first open-source version,[1] save for some extra source code comments, and a version renumbered to 1.3.0-SNAPSHOT. The protocol was described in a draft version of an article, which was ultimately published.[2] This review did not cover any of IronCore's commercial offerings. One consultant performed the engagement, which consisted of 15 person-days of effort, and an additional one-day retest on April 24, 2018.

Scope

Two specific questions were expressed by IronCore:

1. Are the chosen pairing and elliptic curve[3] cryptographically sound and secure?
2. Is the Scala implementation a faithful and correct embodiment of the protocol?

The protocol itself is derived from a 2009 proposal by Wang and Cao; it was however slightly modified by IronCore Labs to make it better fit their use case, in which a single proxy is used for many users and devices, and also to address security weaknesses that were discovered a few years after the original protocol publication. Proxy re-encryption actors (proxies, users, devices, etc.) own public/private key pairs; the binding of public keys to actor identities was not in scope.

Limitations

The underlying Wang-Cao protocol, modified by Cai and Liu, is relatively recent, and has not yet received an extensive academic review. NCC Group's review focuses on whether IronCore's choice of parameters and implementation faithfully follow the protocol, but we cannot guarantee that the protocol itself is secure.

The current IronCore implementation is not "constant-time" and thus may allow secret values to leak through timing-based side channels. This was already known, and a constant-time implementation would require substantial reengineering of the code. Some more information on that subject is detailed in finding NCC-IronCore-recrypt-006 on page 11.

Key Findings

The main findings were the following:

- A key derivation process, to obtain a symmetric encryption key, was identical to a hash value meant for authentication and traveling unprotected on the wire.
- Input curve points were not validated to really belong to the curve, leading to the possibility of small subgroup attacks by using points on alternate curves.

Both issues were fixed in internal version 8.0.1.

Strategic Recommendations

The chosen curve was secure with regards to current technology, but fell short of the expected goal of "128-bit security", at a level of about 100 bits. We recommended switching to a Barreto-Naehrig curve with a larger base field. This recommendation was followed by IronCore Labs. In internal version 11.0.0-SNAPSHOT, a new Barreto-Naehrig curve was defined, with a 480-bit base field, thus ensuring 128-bit security against the best-known discrete logarithm methods in the field extensions used by BN curves.

The current implementation of base field operations uses Scala's standard `scala.math.BigInt` class, which is not optimized for modular arithmetics. We recommend switching to a custom implementation, using Montgomery's multiplication; this would allow better performance, especially when targeting a plain JavaScript output with no WebAssembly support, and is necessary in order to achieve, as a long-term goal, an implementation that is free of timing-based side channels.

Some other improvements on curve operations and Miller's algorithm implementation are detailed in Appendix A on page 13. Internal version 11.0.0-SNAPSHOT includes some of the suggested improvements.

[1] https://github.com/IronCoreLabs/recrypt/commit/4fb521d1c68668f1af5c90ac1993242b96f5a221
[2] Cryptographically Enforced Orthogonal Access Control at Scale, https://dl.acm.org/citation.cfm?id=3201602
[3] New software speed records for cryptographic pairings, https://eprint.iacr.org/2010/186

Dashboard

Target Metadata

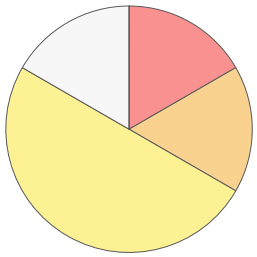
Name	Proxy Re-Encryption Protocol
Type	Library
Platforms	Scala and Scala-js

Engagement Data

Type	Cryptographic and Implementation Review
Method	Architectural and Source Code Review
Dates	2018-02-26 to 2018-03-20
Consultants	1
Level of effort	15 person-days

Finding Breakdown

Critical Risk issues	0
High Risk issues	1
Medium Risk issues	1
Low Risk issues	3
Informational issues	1
Total issues	6



Category Breakdown

Cryptography	4
Data Exposure	1
Data Validation	1

Component Breakdown

Protocol	1
Decryption Engine	1
Algorithm Parameters	1
Schnorr Signature	1
Random Generation	1
Algebraic Primitives	1

Key

Critical | High | Medium | Low | Informational

Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. For an explanation of NCC Group's risk rating and finding categorization, see Appendix B on page 16.

Title	Status	ID	Risk
Derived Keys Are Identical to Authentication Hashes	Fixed	002	High
Input Points Are Not Verified to Belong to the Curve	Fixed	003	Medium
Selected Curve Security Is Lower Than Expected	Fixed	001	Low
Per-Signature Secret Values Are Biased	Reported	004	Low
Random Value Range Is Not Validated	Reported	005	Low
Implementation Is Not Constant-Time	Reported	006	Informational

Finding Details

Finding	**Derived Keys Are Identical to Authentication Hashes**
Risk	High Impact: High, Exploitability: Medium
Identifier	NCC-IronCore-recrypt-002
Status	Fixed
Category	Cryptography
Component	Protocol
Location	• `internal/InternalApi.scala`, methods `encrypt()` and `decrypt()` • `CoreApi.scala`, methods `deriveSymmetricKey()` and `derivePrivateKey()`
Impact	Eavesdroppers may observe the `AuthHash` value, which is sent unencrypted, and is equal to the value of symmetric keys derived from the random plaintext; they may thus decrypt all data encrypted with that key.
Description	The proxy re-encryption protocol handles encryption of plaintext messages which must have a specific format (they are elements of a subgroup of order r of the multiplicative group of invertible elements in the field extension $\mathbb{F}_{p^{12}}$). Since encoding arbitrary data into such a format is complicated and expensive, the plaintext messages are actually random elements, which derived symmetric keys with a SHA-256 hash are derived from, to be used with a classic symmetric encryption algorithm. This process is described in the documentation of the `InternalApi.encrypt()` method:

```
/*
 * Encrypt plaintext to publicKey. This public key encryption is not
 * meant to encrypt arbitrary data; instead, you should generate a
 * random plaintext value (an element of G_T), apply a SHA256 hash to
 * it to generate a 32-bit number, and use that as a key for a
 * symmetric algorithm like AES256-GCM to encrypt the data. Then use
 * this method to encrypt the plaintext.
 */
```

The methods `CoreApi.deriveSymmetricKey()` and `CoreApi.derivePrivateKey()`, which are part of the public API, embody this hashing step.

The encryption protocol also includes computation of a value ah, which is sent as part of the ciphertext (but is not itself encrypted). ah serves as proof that the sender knows the plaintext value; it is defined to be the SHA-256 hash of the plaintext. This is implemented in `InternalApi.encrypt()` and `InternalApi.decrypt()`.

This implies that the ah value, which travels unprotected on the wire, and the derived symmetric key, to be used for further data encryption, are identical. An attacker observing the exchanged traffic can learn the symmetric key, and then easily decrypt all the data.

Recommendation	The authentication hash ah and/or the key derivation process must be modified to ensure that the symmetric key cannot be inferred from the authentication hash.
Client Response	This issue was fixed in version 8.0.1 of the `recrypt` library: the authentication hash ah is now defined to be the hash of the concatenation of the ephemeral public key (also part of the ciphertext) and the plaintext.

Finding	**Input Points Are Not Verified to Belong to the Curve**
Risk	Medium Impact: High, Exploitability: Low
Identifier	NCC-IronCore-recrypt-003
Status	Fixed
Category	Cryptography
Component	Decryption Engine
Location	`internal/point/AffinePoint.scala`, method `apply()` (removed in version 11.0.0-SNAPSHOT)
Impact	Attackers may send carefully crafted invalid messages and observe the recipient's behavior to infer information on the recipient's private key.
Description	An encrypted message is a ciphertext consisting of (among other values) an *ephemeral public key* (**epk**), which is a curve point. That point is sent as two coordinates x and y that should nominally match the curve equation ($y^2 = x^3 + 3$). However, the recipient code does not verify that property. The sender may craft messages where **epk** is on an alternate curve, and specifically on a subgroup of small order of such a curve. For instance, the curve $y^2 = x^3 + 1$ has small subgroups of order 2, 3, 4, 13, 139, 1868033... *Invalid Curve Attacks*[4] use points on small subgroups to obtain some partial information on a private key, usually the remainder of the private key modulo the small subgroup order. The attack requires subgroups small enough for an exhaustive search in that subgroup to be feasible; on the other hand, there are many potential curves, leading to a wide choice of subgroups to work with. In the case of the proxy re-encryption protocol, the attack must go through the pairing computation, and thus depends on how the pairing implementation works when used with invalid inputs; this is a currently unexplored subject.
Recommendation	Incoming points, when decoded, should be systematically verified to belong to the intended curve. The cost of verifying the curve equation (three multiplications in the base field) is negligible with regards to a pairing evaluation or even a generic point multiplication.
Retest Results	Version 8.0.1 of `recrypt` contains a modified `AffinePoint` class that systematically checks the incoming data against the curve equation. In version 11.0.0, the `AffinePoint` class has been removed, and its functionality merged into `HomogeneousPoint`, which now includes the systematic validity check on incoming point data.

[4] https://web-in-security.blogspot.com/2015/09/practical-invalid-curve-attacks.html

Finding	**Selected Curve Security Is Lower Than Expected**
Risk	Low Impact: High, Exploitability: None
Identifier	NCC-IronCore-recrypt-001
Status	Fixed
Category	Cryptography
Component	Algorithm Parameters
Location	Choice of elliptic curve.
Impact	By solving discrete logarithm in the field extension, attackers could recover private keys and decrypt all messages.
Description	The proxy re-encryption protocol requires the use of a pairing-friendly elliptic curve. Such a curve is characterized by the following parameters:

- The base field order (p, usually prime).
- The curve order (n).
- The subgroup order r (a prime number, that divides n).
- The curve embedding degree, which is the smallest integer $k > 0$ such that r divides $p^k - 1$.

The pairing operation transforms the Discrete Logarithm (DL) problem in the base elliptic curve, into a multiplicative DL problem in the extension field $\mathbb{F}_{p^k}$. The achieved "security level" will then be the minimum of the difficulties of DL in the elliptic curve, and DL in the field extension.

The curve chosen by IronCore is the one described by Naehrig, Niederhagen and Schwabe in 2010[5]; this is a specific instance of the Barreto-Naehrig curves.[6] That curve offers the following characteristics:

- Base field order is a 256-bit prime p.
- Curve order n is a 256-bit prime; the subgroup is then the complete curve (i.e. $r = n$).
- The curve embedding degree is $k = 12$.

There is no known DL solving algorithm for elliptic curves with better efficiency than generic group algorithms, which have cost proportional to $\sqrt{r}$. Thus, for elliptic curve DL, this curve achieves "128-bit security", which is the target security level.

For multiplicative DL in the field extension, the traditional analysis is to infer the cost from the best-known generic algorithm for moduli of that size. That algorithm is the General Number Field Sieve. GNFS has two variants, for solving discrete logarithm and for integer factorization; they have the same asymptotic cost, and measures on the current DL record (a 768-bit prime modulus[7]) show that the actual costs are similar. We can thus apply cost estimates for integer factorization on DL in a prime field or field extension of similar size. According to that analysis, the 3072-bit field extension ($\mathbb{F}_{p^{12}}$) should offer multiplicative DL complexity of 124 to 128 bits.

However, it has been recently discovered that this is an overestimate. Indeed, the base field modulus p in a BN curve is expressed as the value of a polynomial $p = P(u) = 36u^4 +$

[5] M. Naehrig, R. Niederhagen, and P. Schwabe, *New software speed records for cryptographic pairings*, https://eprint.iacr.org/2010/186
[6] P. Barreto and M. Naehrig, *Pairing-Friendly Elliptic Curves of Prime Order*, https://eprint.iacr.org/2005/133
[7] T. Kleinjung, C. Diem, A. Lenstra, C. Priplata, and C. Stahlke, *Computation of a 768-bit prime field discrete logarithm*, https://eprint.iacr.org/2017/067

$36u^3 + 24u^2 + 6u + 1$, for a small parameter u (less than 64 bits). This polynomial P has small enough coefficients to allow the use of a Special Number Field Sieve, which is more efficient than GNFS. Barbulescu and Duquesne[8] estimated, for a BN curve with a 256-bit base field and embedding degree $k = 12$, a multiplicative DL cost of a bit less than 2^{100}. The specific curve selected by IronCore uses a different u parameter, but should yield a similar cost (or even a smaller attack cost, since that u happens to be a perfect cube, a property which allows a slight speed-up in computations, but is not a requirement of "normal" BN curves).

The curve selected by IronCore thus offers a security level of only 100 bits or so, possibly less if the special format of the u parameter can be further exploited in SNFS. A 100-bit security level is still beyond current technology; it is roughly as resistant as RSA-2048, which is in common wide usage. It is, however, substantially lower than the expected and targeted "128-bit" security level.

Recommendation

We recommend switching to a larger curve that provides the expected 128-bit security level. Barbulescu and Duquesne suggest new parameters for a BN curve, with a 114-bit u value, yielding a 462-bit base field modulus, and 5544-bit field extension. The currently selected curve is not breakable by current technology, but may induce user distrust by failing to meet the perceived security level of "128 bits" and associated notion of a "security margin".

Retest Results

Version 11.0.0 of the `recrypt` library includes support for a new BN curve with a 480-bit base field size, which is enough to ensure a 128-bit security level. The curve was selected with the same methodology as the Naehrig-Niederhagen-Schwabe BN-256 curve, with a u parameter chosen as a cube of a low Hamming weight integer, with an adequate size to ensure a 480-bit base field. The curve order is prime and has extension degree exactly 12.

[8] R. Barbulescu and S. Duquesne, *Updating key size estimations for pairings*, https://hal.archives-ouvertes.fr/hal-01534101/file/main.pdf

Finding	**Per-Signature Secret Values Are Biased**
Risk	Low Impact: Medium, Exploitability: Low
Identifier	NCC-IronCore-recrypt-004
Status	Reported
Category	Cryptography
Component	Schnorr Signature
Location	`CoreApi.scala`, method `schnorrSign()`
Impact	Observation of many signatures could theoretically allow reconstruction of the signature private key.

Description

A Schnorr signature involves creation of a random private non-zero element k modulo the curve order n. A new k is generated for each signature value. In the context of the DSA and ECDSA signature algorithms, an unpublished attack by D. Bleichenbacher shows how the signature private key can be recovered from analysis of many signature values if the per-signature element generation process is biased. If the bias is as little as 3 known bits in the value of k, then the private key may be recovered with only a hundred signature values, as explained by Nguyen and Shparlinski in 2003.[9] If the bias is smaller, it can still be exploited, but with a rapidly increasing number of required signatures.

In the `recrypt` library, the value of k is obtained by generating a sequence of 256 random bits, interpreting it as an integer, and reducing it modulo the curve order. Since the curve order n is close to $0.561 \cdot 2^{256}$, values below $0.439 \cdot 2^{256}$ are twice as likely to be selected than values above that threshold. Since all values are still possible, the bias is much lower than in the Nguyen and Shparlinski analysis. In a 2001 report from S. Vaudenay[10] (section 5), some extra information is provided on Bleichenbacher's attack, stating that such a small bias could still be exploited with 2^{22} signatures, and a substantial computational cost (2^{63} operations).

The intended use of Schnorr signatures in the IronCore systems is to compute a single signature as part of a device initial configuration. As such, it is expected that there will never be a sufficient number of available signature values, for a given private key, to allow the attack to apply. Nevertheless, Vaudenay states that the attack might be made more efficient, and very little has been published on that subject since.

Recommendation

We recommend amending the per-signature random generation process in order to make it uniform. Two main strategies are available:

1. Generate a random value of exactly 256 bits, but reject it if it does not fall in the proper $[1..n-1]$ range, instead of using a modular reduction. This involves using a loop to repeat the process until it succeeds; the loop is already present in the code, to handle rare cases where the resulting curve point turns out to be the point at infinity.
2. Generate a random value substantially longer than 256 bits, and apply a modular reduction. It is easily shown that doubling the length, i.e. generating a random 512-bit value, is more than enough to make any bias negligible.

[9] P. Nguyen and I. Shparlinski, *The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces*, https://pdfs.semanticscholar.org/0eb1/8a42b623dd8e7cdd4221085a6fd5503708ea.pdf

[10] S. Vaudenay, *Evaluation Report on DSA*, https://www.ipa.go.jp/security/enc/CRYPTREC/fy15/doc/1002_reportDSA.pdf

Finding	**Random Value Range Is Not Validated**
Risk	Low Impact: Medium, Exploitability: None
Identifier	NCC-IronCore-recrypt-005
Status	Reported
Category	Data Validation
Component	Random Generation
Location	`CoreApi`
Impact	An improper random source behavior would fail to be detected, and then lead to breaches in confidentiality of data.
Description	Random values must be generated for all key generation and encryption operations. Current `recrypt` implementation takes an `IO[ByteVector]` object as one of its parameters, and this is supposed to produce sequences of random bytes on demand. The *documentation* states that the object "should return a new cryptographically random `ByteVector` that is at least 32 bytes long on each invocation." Whenever a random value is needed, either modulo the base field order (p) or modulo the curve order (r), a single `ByteVector` instance is obtained, interpreted as a big integer (with unsigned big-endian convention), and reduced modulo the relevant value (p or r).[11] Nothing in the implementation verifies that the source value has a length of at least 32 bytes. If the externally provided implementation returns shorter values, then this is equivalent to setting upper bits of the private value to zero, correspondingly reducing the cryptographic strength. For instance, if the random `ByteVector` is only 16 bytes long, then generic discrete logarithm algorithms will recover the private elements, and decrypt data, with cost only 2^{64}. Such an issue would go undetected, because biased random generation does not have any *functional* consequences. Moreover, when a larger curve is used, the "32 bytes" would not be enough.
Recommendation	The provided `IO[ByteVector]` object should be invoked only by a dedicated function that will check that the length of the returned sequence of bytes fulfills the security requirements; that function may also invoke the source repeatedly and concatenate the obtained chunks until the target length is reached. Extra care should be applied to the following points: • The number of random bytes depends on the target modulus. Larger curves need more. • To ensure uniform generation of values modulo an integer m of e bits (i.e. $2^{e-1} \leq m < 2^e$), the generator should get exactly e random bits (possibly accumulating or truncating the obtained random bytes), then interpret the random bits as an integer; if the resulting value is not in the proper range, then it should be rejected, and the process restarted. • The generation function should be parameterized with the proper range, e.g. "1 to $p - 1$" for a random non-zero field element.

[11] In the case of random values modulo the curve order, the reduction is implicit in the point multiplication algorithm.

Finding	**Implementation Is Not Constant-Time**
Risk	Informational Impact: Medium, Exploitability: Undetermined
Identifier	NCC-IronCore-recrypt-006
Status	Reported
Category	Data Exposure
Component	Algebraic Primitives
Location	`Fp`, `HomogeneousPoint`
Impact	Side-channel attacks may reveal parts of private keys and secret data.

Description

The current `recrypt` implementation is not "constant-time," which means that timing attacks may recover some secret data elements. Generally speaking, timing attacks exploit the following types of operations:

- Elementary operations with a data-dependent execution time (e.g. integer divisions).
- Data-dependent conditional jumps.
- Memory accesses at addresses that depend on secret data.

These last two elements can be exploited through cache attacks, which measure cache content status through timing.

In the current `recrypt` implementation, the following elements in particular are not constant-time:

- **BigInt**: Scala's standard implementation of big integers (which is Java's `BigInteger` when running on the JVM) truncates values to their minimal mathematical length, and contains many conditional jumps to handle special cases (e.g. zero).
- **Curve operations**: classic curve point doubling addition formulas include special cases:
  - When one of the operands is the "point at infinity"
  - When adding two points that are opposite to each other
  - When adding two identical points

  Management of such cases uses conditional jumps that depend on secret data.
- **NAF**: When multiplying a curve point by a scalar, `recrypt` uses a double-and-add algorithm optimized through the use of a non-adjacent form, which allows making only one curve point addition for every two doubling operations. Unfortunately, the use of a NAF necessarily leaks information about the scalar, since additions will occur at data-dependent moments.

Whether timing attacks are feasible depends on the library usage context. Most demonstrations of timing attacks have been performed locally, i.e. from attacker-controlled code running on the same hardware (possibly from another virtual machine running on a distinct core on the same CPU). However, in lab conditions, remote timing attacks have been demonstrated.[12]

Recommendation

As a long-term goal, we recommend modifying the `recrypt` implementation to become constant-time. This will require the following:

1. Replace the implementation of modular arithmetics (`Fp` class, and `BigInt`) with a dedi-

[12] D. Boneh and D. Brumley, *Remote timing attacks are practical*, https://crypto.stanford.edu/~dabo/abstracts/ssl-timing.html

cated class that would use Montgomery's multiplication.
2. Switch curve operations to constant-time formulas. In particular, "complete" formulas[13] remove the need for special case management code.
3. Replace NAF with a window-based optimization (with a constant-time array lookup) when multiplying a curve point with a secret scalar value. Note that the use of NAF in Miller's algorithm is not a problem, since in that case the non-adjacent form is over a fixed public value.

[13] See for instance: https://eprint.iacr.org/2015/1060

Appendix A: Optimization Notes

This section describes various possible optimization strategies to help with secure integration and performance of the library.

Operations on the Base Field

Current `recrypt` implementation of operations in the base field $\mathbb{F}_p$ is based on the standard `scala.math.BigInt` from the Scala library, which itself wraps around `java.math.BigInteger`. In Oracle's JDK, `BigInteger` is implemented in pure Java; in Scala-js, a pure Scala implementation[14] is used. Both are based on the same structure: big integers are represented as a sign, and a positive mantissa as a base-2^{32} value (each "digit" is traditionally called a "limb"), which is contained in an array of 32-bit integers. These big-integer implementations provide little support for *modular* computations, though.

Multiplications

`recrypt`'s `internal.Fp` class performs multiplications by applying the operation on the underlying big integers, then reducing modulo the field order p with the remainder operator (`%`). This internally triggers the generic integer division algorithm.

A faster implementation strategy would be to use Montgomery multiplication. If a modular integer is represented as an array of n limbs in base W (e.g. $n = 8$ and $W = 2^{32}$, for a 256-bit integer represented as an array of 32-bit integers), then we define the value $R = W^n \mod p$. The *Montgomery representation* of a modular integer x is equal to $xR \mod p$. The *Montgomery product* of x by y is $xy/R \mod p$. The Montgomery product of the Montgomery representation of two integers is then the Montgomery representation of the product of the two integers:

$$(xR)(yR)/R = (xy)/R \mod p$$

Converting to and from Montgomery representation is easily done by applying a Montgomery product with the constants, respectively, $R^2 \mod p$ and $1 \mod p$.

The implementation of the Montgomery product can be described as two regular intertwined multiplication loops. Notably, it is amenable to constant-time implementations and efficient loop-unrolling, since most of the algorithm jumps are independent of the processed values. For details on the Montogmery product, see chapter 14, especially algorithm 14.36, of the Handbook of Applied Cryptography.[15]

The `BigInt.modPow()` method internally uses Montgomery multiplication. This allows a quick benchmark:

```
val e = java.math.BigInteger.ONE.shiftLeft(10000);
for (a <- 1 to 5) {
    var begin = System.currentTimeMillis();
    for (b <- 1 to 100) {
        x = x.modPow(e, p)
    }
    var end = System.currentTimeMillis();
    println(s"modPow: ${end - begin}")
    begin = System.currentTimeMillis();
    for (b <- 1 to 1000000) {
        y = Fp(y * y)
    }
    end = System.currentTimeMillis();
    println(s"Fp mul: ${end - begin}")
}
println(s"x = ${x}, y = ${y}")
```

Using OpenJDK 1.8.0_151 on an 64-bit x86 system, this benchmark shows the `modPow()` method (hence Montgomery multiplication) to be about twice as fast as the implementation used in `recrypt`.

[14] https://github.com/scala-js/scala-js/blob/master/javalib/src/main/scala/java/math/BigInteger.scala
[15] The Handbook is freely downloadable at: http://cacr.uwaterloo.ca/hac/

Inversion

Modular inversion is performed in `recrypt` using a custom extended Euclidean algorithm implementation (in the `internal/package.scala` file, method `gcde()`). However, `scala.math.BigInt` offers a dedicated `modInverse()` method. In Oracle's JDK, this maps to a *binary* GCD implementation, which is both asymptotically and practically faster. A simple benchmarck indicates that `modInverse()` is about four times faster than `recrypt`'s implementation.

Even though inversion remains an expensive operation, with a cost of about 50 to 100 times that of a modular multiplication, it is important for performance to avoid inversion when possible. The use of projective or Jacobian coordinates for elliptic curve points, for instance, allows reducing the number of required inversions.

If a constant-time implementation is required, then inversion of x modulo p can be performed with a modular exponentiation (with exponent $e = p - 2$) that can be implemented in a constant-time way. For a 256-bit modulus, such an exponentiation will cost about 300 multiplications, thus slower than a binary GCD, but still not intolerably slower if the overall algorithm took care to reduce the total number of inversions.

Limb Size

Both `java.math.BigInteger` (JVM) and `scala.math.BigInt` (Scala-js implementation) use 32-bit limbs. This is not necessarily optimal.

WebAssembly offers explicit 32-bit and 64-bit integer types. The multiplication on 64-bit integers yields only the low 64 bits of the result; the high 64 bits are inaccessible, even though the underlying hardware may be able to compute them (at least on 64-bit architectures). Thus, limbs shall be no more than 32 bits in size, so that intermediate products are not truncated.

However, a slightly smaller limb size may yield better performance. When using limbs of t bits, the inner loop of a Montgomery product uses intermediate values of $2t + 1$ bits in size; if using 32-bit limbs, then these values have a size of 65 bits, which does not fit into a single variable or register. Instead, the operations must be broken into two separate carry propagation loops, which impacts performance. Experiments on C implementations show that using 31-bit limbs promotes performance, making multiplications up to twice as fast. The extra bit in each limb is also convenient for expressing carry propagation in constant-time code.

Plain JavaScript, however, has a unique arithmetic type, which is double-precision floating point numbers. JavaScript JIT compilers try to map operations to 32-bit integers when possible, but this cannot be done in all generality for multiplications: the multiplication of two 32-bit integers yields a 64-bit value that exceeds the 53 bits of precision allowed by IEEE 754 double-precision values. JavaScript semantics therefore mandate rounding, which does not yield the same results as truncation to the low 32 bits. For that reason, asm.js, a predecessor to WebAssembly, mandated a `Math.imul` function to provide the low 32 bits of a product of two 32-bit integers. Even if present on a specific implementation, though, `Math.imul` does not help with big integers.

If targeting a pure JavaScript output, then a better implementation would use 26-bit limbs, encoded as floating-point values. Specifically, a limb of value w (with $0 \leq w < 2^{26}$) would be encoded as a double-precision value $2^{-13}w$. A product of two such values yields a result that can be represented with no loss of information; moreover, the "high" half of the result (26 bits) can be obtained with an efficient truncating conversion to integers (`cvttsd2si` opcode in x86 assembly).

Elliptic Curve Point Additions

The `AffinePoint` and `HomogeneousPoint` classes implement operations on elliptic curve points.[16] `AffinePoint` is a straightforward application of the curve formulas, thus requiring a modular division, which is expensive. `HomogeneousPoint` uses projective coordinates to avoid modular inversions (one inversion will still be needed at some point, but possibly after many point additions).

[16] The `AffinePoint` class was removed from version 11.0.0-SNAPSHOT.

The formulas used in `HomogeneousPoint` entail the following costs:

- `double()`: 12 multiplications
- `add()`: 20 multiplications (including 4 multiplications for an initial comparison, to use `double()` instead when required)

Some of these multiplications are squarings. These counts do not include multiplications by small integers, since these can be implemented more efficiently than generic products.

Better formulas exist. Classic implementations use *Jacobian coordinates*, in which a point (x, y) is represented by a triplet $(X : Y : Z)$, which is such that $x = X/Z^2$ and $y = Y/Z^3$. In such a representation, the costs of `double()` and `add()` are 8 and 16 multiplications, respectively.[17] Mixed-coordinates addition, when one of the operands is in affine coordinates (i.e. $Z = 1$) is only 11 multiplications. Moreover, the equality test to fall back to doubling can be integrated in the evaluation with negligible cost.

Complete formulas have later been found by Marc Joye,[18] and then even better formulas by Joost Renes, Craig Castello and Lejla Batina.[19] These formulas work over projective coordinates and have a cost of 12 multiplications for a curve of equation $y^2 = x^3 + b$ with a small constant b (11 multiplications for mixed-coordinates addition). These are *complete* formulas, that have no special case,[20] and thus also work when the two inputs are the same point, or are two opposite points. There are nevertheless specialized doubling formulas that require only 8 multiplications, a cost similar to that offered by Jacobian coordinates.

Use of the Renes-Castello-Batina formulas would provide substantial performance improvements over the current implementation in `recrypt`, while at the same time removing special cases, which would make the implementation simpler and more amenable to a future constant-time implementation.

Miller's Algorithm

Miller's Algorithm powers the first half of the Ate pairing computation, an essential and computationally expensive part of the proxy re-encryption protocol. Conceptually, it consists in an evaluation of a high-degree rational function over a curve point. That function can be expressed as a product of simple rational functions that correspond to lines on the plane. Evaluation will consist of $\log r$ "doublings" (function `InternalApi.doubleLineEval()` in `recrypt`) and about $(\log r)/2$ "adds" (`InternalApi.addLineEval()`), following a non-adjacent form representation of the base curve order r.

Each of `doubleLineEval()` and `addLineEval()`, in its current form, implies two inversions in $\mathbb{F}_{p^2}$. Each inversion requires one inversion in $\mathbb{F}_p$. Since such operations are very expensive (cost is similar to 50 to 100 multiplications), these inversions represent the vast majority of the current implementation cost.

A more efficient strategy would use rational numbers. Since the values obtained are simply multiplied together, each could be represented as a fraction numerator/denominator. The numerators and denominators would then be multiplied together. Only at the end of Miller's algorithm would a single inversion (in $\mathbb{F}_{p^{12}}$) be needed.

Retest result: the representation of intermediate values as fractions, as suggested above, has been implemented in version 11.0.0-SNAPSHOT.

[17] Cost is 9 multiplications for doubling in general; however, in curve $y^2 = x^3 + ax + b$, one multiplication can be avoided if $a = 0$ or $a = -3$.
[18] M. Joye, *Complete Addition Formulæ for Elliptic Curves*, http://joye.site88.net/techreps/complete.pdf
[19] J. Renes, C. Castello, and L. Batina, *Complete addition formulas for prime order elliptic curves*, https://eprint.iacr.org/2015/1060
[20] The formulas have one case that they do not support correctly, which is when the difference of the two operands is a point of order 2. However, when working on the r-torsion subgroup of a curve, with r an odd prime, there are no points of order 2, and this case does not happen in practice.

Appendix B: Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Critical Implies an immediate, easily accessible threat of total compromise.

High Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.

Medium A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.

Low Implies a relatively minor threat to the application.

Informational No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

High Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.

Medium Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.

Low Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

High Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.

Medium Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.

Low Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.